AD-A018 801

SAMSO COMPUTER LANGUAGE AND SOFTWARE DEVELOPMENT ENVIRONMENT REQUIREMENTS

E. D. Callender, et al

Aerospace Corporation

Prepared for:

Space and Missile Systems Organization

15 December 1975

006063

REPORT SAMSO-TR-75-290

SAMSO Computer Language and Software Development Environment Requirements

E. D. CALLENDER, M. FELICIANO, and L. D. JENNINGS
Information Processing Divisio n
Engineering Science Operations
The Aerospace Corporation
El Segundo, Calif. 90245

15 December 1975

Final Report

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

Prepared for

SPACE AND MISSILE SYSTEMS ORGANIZATION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
Los Angeles, Calif. 90045

57

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SAMSO-TR-75-290	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SAMSO Computer Language and Software Development Environment Requirements		5. TYPE OF REPORT & PERIOD COVERED Final Report 4-15-75 to 6-1-75
		6. PERFORMING ORG. REPORT NUMBER TR-0076(6112)-2
7. AUTHOR(s) E. D. Callender M. Feliciano L. D. Jennings		8. CONTRACT OR GRANT NUMBER(s) F04701-75-C-0076
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Aerospace Corporation El Segundo, California		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Space and Missile Systems Organization Air Force Systems Command Los Angeles, California 90045		12. REPORT DATE 15 December 1975
		13. NUMBER OF PAGES 57
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Language
Software Development Environment
Language Constructs
Implementation Constructs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this report is to identify the higher order language requirements and software development environment requirements for SAMSO applications. This report is prepared in response to AFSC Program Directive E215-1-75-30 on Higher Order Language (HOL) Standardization for Computer Resources in Systems. Three major sections constitute this document: (1) functional requirements for SAMSO computer programs, (2) higher order language constructs and software development environment

DD FORM 1473 (FACSIMILE)

19. KEY WORDS (Continued)
Compiler
Operating System
Software Testers
Software Qualifiers
Coding Standards
Coding Standard Enforcer

20. ABSTRACT (Continued)

constructs necessary to support the functional requirements, and (3) recommendations. It is technically feasible for the vast majority of all computer programming for new SAMSO projects to be done in one higher order language. If this technical feasibility is coupled with the increasing cost of software development and maintenance, standardization on a single higher order language becomes highly desirable. If cost savings in software development and maintenance are to be realized, language standardization is only part of the issue. The other main features that must be attacked and successfully resolved are the creation of a standard software development environment and the establishment of a program for the specifications, development, test, and maintenance of this single higher order language and its associated software development environment.

Approved by

L Sashkin

L. Sashkin, Director
Data Processing
Subdivision
Information Processing Division
Engineering Science Operations

R O Bock

R.O. Bock, Group Director
Guidance and Computer Group
Directorate
Technology Division
Development Operations

Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

[illegible]

FRANK P. DYKE, Lt Col, USAF
Chief, Computer Technology Division
Deputy for Technology

PREFACE

This report was prepared for the Technology Division Guidance and Computer program office of The Aerospace Corporation by members of the Information Processing Division. The authors gratefully acknowledge substantial contributions and helpful suggestions to this report on the part of B. A. Corn, H. Hecht, D. J. Reifer and Major C. S. Jund (SAMSO/DYAC).

CONTENTS

CONTENTS (Continued)

FIGURES

TABLES

1. INTRODUCTION

This report has been prepared at the request of the Computer Directorate, Technology Division. It ascertains the SAMSO higher order language and software development requirements. The need for this information arose from AFSC Program Directive E215-1-75-30 on Higher Order Language (HOL) Standardization for Computer Resources in Systems, and from a report of the Department of Defense working group on "Strawman HOL Requirements." The report is based on the authors' background in the requirements of the various cited projects. The short time available for the preparation did not permit formal canvassing of SAMSO program offices.

Three principal sections are provided herein: (1) functional requirements for SAMSO computer programs, (2) higher order language constructs and software development environment constructs necessary to support the functional requirements, and (3) recommendations. Appendix A deals with a detailed review of the SAMSO functional requirements for computer programs; Appendix B concerns an assessment of compiler writing technology; Appendix C provides an outline of a plan for specifying such a language and its associated implementations.

SAMSO requires the capability to obtain high quality computer programs at reasonable cost for a wide variety of space and missile programs. Once obtained, these computer programs must be maintained and an upgrade of functional requirements often imposes a requirement for modifications in the supporting software. Costs to support such software cycles exist in an environment where there has been a dramatic shift in the relative cost of computer hardware and software. The shift has been from hardware costs being dominant to software costs being dominant. Also, Air Force programs are increasing in functional complexity and usually rely on a rapidly changing technology. These changes have made the production of software an extremely important and costly activity. One step that can be taken to ease the spiraling costs is to standardize, where possible, on a higher order language and associated software development environment. This report is a review of the requirements in connection with such a standardization.

2. SAMSO MISSION

SAMSO encompasses seven major System Program Offices (SPO's), the Space and Missile Test and Evaluation Center (SAMTEC), and the Air Force Satellite Control Facility (AFSCF). Each of these deputy organizations uses software in several of the following categories:

- Operational Flight Programs
- Range Safety and Control
- Communications
- Command and Control
- Simulation and Training
- Automatic Test Equipment
- Support software for each of the above categories

This list of categories should not be considered exhaustive. It is included to display the diversity of applications presently employing software as a major systems component. SAMSO's diverse applications range from operational flight programs for small, militarized computers (e.g., Minuteman) to large-scale command and control (e.g., AFSCF). Appendix A contains a breakout of the functional requirements for computer programs. However, to assist in determining the impact of such functional requirements upon computer language requirements, a classification scheme that reflects the computer hardware and software constraints found in practice is used. This classification scheme, used below, is different than the one used in Appendix A to determine functional categories.

In terms of language requirements, the software usage can be classified into:

- Flight Programs
- Real-Time Responsive Programs
- Other Ground Programs

Later comments are organized on the basis of this classification. It is characteristic of the SAMSO mission that computer programs are developed, frequently independently tested, and usually maintained by contractor organizations, drawing personnel from the general software labor pool. As a consequence, language features that are at variance with commercially used programming practices may constitute a handicap.

3. SAMSO COMPUTER PROGRAM REQUIREMENTS

To analyze computer language requirements, software programs will be placed into one of the following three categories: (1) on-board flight programs, (2) real-time responsive software support programs, and (3) non-real-time programs. Programming of the first type is characterized by an environment of limited size, high reliability requirements, and possible hostile or system degrading effects. Usually the amount of computer storage and processing capability is extremely limited. A very reliable software system is required since software malfunctions may result in mission degradation or failure. Also environmental factors such as nuclear effects that may cause hardware and software malfunctions must be taken into account.

The remaining two categories of programs assume that the hardware system is on the ground and is under the physical control of friendly personnel. Usually the hardware resources are sized such that if efficient use is made of them, the necessary task can be accomplished in the desired period of time. Real-time responsive computer software systems include such things as real-time ground support, ground computations for radio control guidance schemes, real-time range safety work, and certain portions of data reduction activities where the data obtained from an analog source must be suitably processed in real-time to prevent it from being lost.

4. RECENT SAMSO EXPERIENCES USING HIGHER ORDER LANGUAGE IN OPERATIONAL PROGRAMS

SAMSO has had over 10 years of experience in the use of higher order languages in operational programs. One of the first large operational systems to use a higher order language was the AFSCF. They have found JOVIAL (J4) to be a very useful language. JOVIAL (J3B) and FORTRAN will be used in the Small Processing Station Program for the Defense Support Program. The Global Positioning Satellite Program has decreed that all computer programs will be written in FORTRAN. For many years various portions of support and automatic test and evaluation software have been written in FORTRAN for a number of different operational programs.

SAMSO and The Aerospace Corporation have been actively involved for the past 5 years in the development and application of a major software system designed to facilitate the production of compilers of higher order programming languages for the minicomputers used in space, missile, and avionics systems. The compilers developed for these real-time processors must satisfy stringent requirements for the production of executable code that is highly efficient both in memory utilization and in execution speed. The methodology must also be cost-effective to use, since the wide variety of digital processors available for weapons systems applications requires a multiplicity of compilers.

The basic system, Space Programming Language Implementation Tool (SPLIT), was used successfully to develop compilers that accept the source language, Space Programming Language, and generate executable code for the Honeywell DDP-516 and RCA SCP-234 computers. The first was used to implement the flight program and ground laboratory support software required for the Space Precision Attitude Reference System; the second to implement the flight program for the Defense Meteorological Satellite. Each compiler was produced in 10 man-months and delivered in 5 calendar months which demonstrated the effectiveness of the methodology in meeting all efficiency goals in a cost-effective manner.

Following completion of these programs, Aerospace began a 2-year program sponsored by the SAMSO Deputy for Technology to improve and enhance the basic system. Because no standard programming language currently exists for the general class of real-time computational systems, SAMSO/ Aerospace felt it was important that the system be enhanced and expanded to readily accept any of the various languages now in use for such systems. It was also deemed necessary to improve the operational efficiency of the system to minimize the cost of its use.

5. SAMSO SOFTWARE CONSTRUCT REQUIREMENTS

5.1 INTRODUCTION

The intent of this section is to briefly delineate the language[*] constructs and software development environment constructs that are necessary to support SAMSO software development activities. The intent is to provide a set of language constructs that will support a stable, well defined environment for most applications (note "most" as opposed to "all"). Emphasis will be on usability; that is, sacrifices of clarity and ease of use will not be made in the name of efficiency. However, such conflicts are deemed to be rare.

There should be an axiomatic definition of the syntax of the language in the sense that a formal grammar for it should exist. The grammar should be context free and it should not contain extraneous elements. In the terminology of the Backus-Naur form, it should not contain useless productions. The syntax should be unambiguous and it should be provable that this is the case. Also, the syntax should be consistent. Moreover, the constructs of the language should be free of ad hoc restrictions. The semantics of the language should be determinable from the description. A reasonable interpretation of a construct should be the only interpretation for that construct. There should not be a facility that allows extension of the syntax. The syntax should be based on conventional forms. Spaces should be used as token separators and a delimiter should be used to separate or terminate statements. The key words should be short and mnemonic.

As the syntax and semantics for such a language are being prepared, the following goals should be kept in mind. The language should support the notion of self-documentation at the code module level. This does not mean

[*] In the remainder of this report, when the term "the language" is used, it refers to the higher order programming language under discussion as the main topic of this report.

that any coding which is done will automatically be self-documenting, but rather that with good coding discipline, self-documentation can be achieved. Another goal is that it should be relatively easy to train knowledgeable programmers in the use of the language constructs and software development environment constructs. To properly interpret this goal it should be remembered that the language is being designed for use by personnel experienced in the development of highly complex software projects. To achieve these goals and produce implementations that will result in efficient execution-time code requires a delicate balance between the demands of the users, the current software technology, and the resourcefulness of compiler and system writers.

Of equal importance to the set of language specifications is the set of specifications that define the software development environment. Here, too, the specifications are designed to cover most applications. Special features will not be included for rare cases and the specifications will be based upon software practices that are well within the state of the art. In the specifications for both language and software development environments there are features which reflect the stringent requirements for verification and validation of software that is to be used in operational SAMSO programs. For example, all data elements must be explicitly declared and initialized. The initialization may be to a default value or done explicitly by the coder.

5.2 LANGUAGE CONSTRUCTS

This section contains an outline of language constructs necessary to support SAMSO computer programming. By intent, this section is not a set of specifications for language constructs; it is merely a listing of those constructs which are deemed necessary with sufficient descriptive material such that a language designer could determine the intent of the requester if he were to prepare the detailed specifications for such a language. Constructs for such a programming language should cover the following seven areas:

- Program Elements
- Data
- Operators
- Structure of Programs
- Control Structures
- Input/Output
- Direct Code

The extent of the constructs in any particular area determine the ease with which an application may be implemented. If a particular construct is not available, the programmer can affect its usage by additional programming or, if necessary, by relying on code written in assembly or machine language.

5.2.1 Program Elements

The two basic ingredients from which programs are created are an alphabetic set and basic program elements. The alphabet of the language should be specified with rules that will allow for an unambiguous implementation of the alphabet in a 64-character set. The basic elements of the language are built from the alphabet and constitute the primitives from which the syntax is established. Basic elements include identifiers, key words, statement delimiters, and separators. These elements are used in the construction of simple statements and groups of statements making up a block of program statements. The basic elements of the language should be specified unambiguously and it should be possible to construct an efficient program to recognize the basic elements used in a source program. This can be achieved, for example, by defining the primitives as an extended regular expression over the alphabet. Sufficient flexibility and richness of the primitives should exist to allow for naturalness and ease of use. For example, it is essential that alphanumeric characters of a reasonably large number be allowed for labels such that meaningful names can be used.

5.2.2 Data

The constructs necessary to allow the programmer to handle a wide variety of data fall into five classifications: (1) data type, (2) aggregation of data elements, (3) initialization of data elements, (4) scope of data, and (5) data storage allocation (and storage control).

Data may exist as either constants or variables. The constructs in the language must be able to handle the following:

- Numeric
  - Integer
  - Fixed Point
  - Floating Point
- Logical
  - Boolean
- Textual
  - Character
  - Bit
- Location
- Pointers

Numeric elements will be provided with multiple precisioning. The language constructs will not provide the capability for new data types. Table I compares the requirements for various data types versus types of computer programs.

To aid in verification and validation all data elements must be explicitly declared both with respect to type and initial value. The initial value is determined at compile time. Further, it is felt that there does exist a requirement to have a "RESET" capability for reinitialization of data during execution time, for example, to support the operating systems that must be written for the special purpose flight computers.

To support the ability to manipulate a large amount of data (with a few statements), it is necessary to be able to aggregate data elements in meaningful ways. The constructs must support the notions of (1) a scalar, that is, a single data element, (2) an array of data elements which consists of a homogeneous set of data elements each of which can be referenced by using the array name and a subscript, and (3) some form of aggregation of a nonhomogeneous set of data elements (see Table II). An example of this latter type is a table which is a set of arrays where the individual arrays can be different in size and type. Although it is clearly possible to have the constructs for a number of other forms of data aggregation for SAMSO applications, such constructs are not necessary for most of the required applications and, hence, they should not be included in the required constructs for the language.

It is necessary to be able to initialize data. Constructs should be available that will allow this initialization to be done at compile time and during execution. The ability to do such initialization should support all data types and all forms of data aggregation.

It is necessary to have constructs in the language that support the declaration of the scope of a data element. The scope will be fixed at compile time, that is, it is statically--not dynamically determined. The language constructs will allow the scope of a data element to be restricted to only those modules where it is intended to be used. However, the capability should exist to identify data within a particular scope that is exportable to other scopes. This should include separately compiled scopes. There should be a hierarchy for scopes that vary (under user control) from local through global. In addition, there should be other scopes such as "inaccessible" which are under control of the compiler and operating system. An example of the application of inaccessible data are the parameterization by the compiler of data representations that are machine dependent; for example, the number of bits in a character and the particular internal representation for the externally uniform character set.

Table I. Data Types

	Flight	Real Time Ground	Other Ground
Numeric			
Integer	X	X	X
Fixed Point	X	X	X
Floating Point	X	X	X
Multiple Precision	X	X	X
Logical			
Boolean	X	X	X
Textual			
Character		X	X
Bit	X	X	X
Location	X	X	X
Pointers	X	X	X

Table II. Data Groups

	Flight	Real Time Ground	Other Ground
Array			
1 and 2 Dimensional	X		
n Dimensional		X	X
Table		X	X

There should be constructs in the language that allow the user, if he so wishes, to control data storage allocation. This includes the ability for dynamic allocation and deallocation of storage areas. The specifications must clearly establish the rules for static and dynamic scope of allocation. Implicit storage allocation will be provided as the default option.

5.2.3 Operators

One of the major functions of a computer program is to appropriately modify a set of data elements. Such modification is done through a variety of operators. The order of evaluation of operators must be clear. The type of operators that must be available at the data element level are arithmetic and relational operators for both scalar and nonscalar data, logical operators (and, or, not, exclusive or) and textual and string operators (concatenation, substring and length). There must be a capability within the language to handle fixed point arithmetic and the associated scale factors (see Table III). Careful consideration needs to be given to type conversion operations. For example, one needs to be able to automatically convert from fixed to floating representation. However type conversions, such as the following example, often lead to implementation ambiguities and should be avoided.

Example:

```
    Real x, y;
    String a;
    Read (a);
    x: = a + y;
```

(The above example is only given for the purpose of illustration. It is not intended to imply specifications for the language.)

5.2.4 Structure of Programs

The experience of the past few years has indicated the wisdom of providing language constructs that would support what is commonly called structured programming. A more descriptive term is structured coding for control of program flow. The fundamental construct here is the notion of a

Table III. Operators

	Flight	Real Time Ground	Other Ground
Arithmetic			
Scalar	X	X	X
Nonscalar	X	X	X
Relational/Boolean	X	X	X
Textual/String		X	X

Table IV. Structure of Programs

	Flight	Real Time Ground	Other Ground
Block Structure	X	X	X
Main Program	X	X	X
Sub Program Types			
Internal Procedures	X	X	X
In-Line Code	X	X	X
Subroutines	X	X	X
Reentrant	X	X	X
Recursive			X
Built-in Functions and User Defined Functions	X	X	X
Data Control Between Sub-Programs	X	X	X
Control of Variable Storage	X	X	X

block of code. The language should have the ability to indicate the "BEGINNING" and "END" for any block of code. Further, every "END" must match exactly one "BEGIN."

The notion of blocks of code extends upward. The language should allow programs to be structured into a hierarchy of subprogram types (see Table IV). At the top of this hierarchy is a main program, followed by independently compiled subroutines. Within a particular subroutine there are different types of subroutine dependent subprograms. An example of these are internal procedures and function statements. It is necessary to be able to efficiently direct the control of program flow from one module to another. In addition, it is necessary to have constructs within the language that allow data to be shared across the boundaries of the hierarchal program structure. Just as there is a hierarchy in program control structure, there should be a hierarchy in the ability to share data. The notion of compool, global versus local variable and additional constructs such as "application inaccessible" need to be utilized.

5.2.5 Control Structures

There are four types of control structures. There are constructs to support (1) transfers, (2) conditional structures, (3) iteration, and (4) multi-tasking. It is mandatory that the language be able to support nesting of different forms of control statements (see Table V). There must be language constructs to support (1) conditional and unconditional transfers, (2) events or interrupts and (3) switches. The best example of a construct for transfer is the famous "GO TO." The language construct for event or interrupt will support a real-time clock and the notion of a real-time event. The results of all possible states will be explicitly stated for all control structures. The existence of side effects is recognized and will be addressed in the specifications. The types of constructs necessary to support a structured coding environment in the area of conditionals include the following types of statements: (1) if, (2) if then else, (3) case (a generalized"if then else"statement). To support iteration there should be the following types of

Table V. Control Statements

	Flight	Real Time Ground	Other Ground
Transfer			
GO-TO	X	X	X
On Interrupt or Event	X	X	
Switch	X	X	X
Conditional			
If	X	X	X
If Then Else	X	X	X
Case	X	X	X
Iteration			
Do	X	X	X
While	X	X	X
Until	X	X	X
Multi-tasking	X	X	

statements: (1) do, (2) while, (3) until and (4) a mechanism to "escape" from the middle of an iteration loop. To support multi-tasking or parallel processing capabilities should be included for (1) creation, (2) activation, (3) synchronization,and (4) termination of the processes.

5.2.6 Input-Output

Input and output operations are required for the computer to communicate. Such statements can be divided into two classes: declarative statements that describe the file, the devices, and the data formats and imperative statements that are used to effect the actual I/O operation and to control the I/O device. The I/O language constructs for flight programs can be relatively simple since the number of on-board devices is very limited. However, for all ground applications the constructs must be extremely broad to allow one to access the wide proliferation of available devices and data formats. There should be the ability to dynamically assign and reassign I/O devices. The types of permitted declarative statements should support both formatted and unformatted record transmission. For example, the requirement is almost nonexistent for formatted record transmission within a flight program while it is in actual operation. However, during program development, verification and validation, and automatic test and evaluation of such a program, there exists a requirement for formatted record transmission.

5.2.7 The Need for Direct Code

For any project which requires the use of a computer system of limited capabilities it is most probable that there will be a requirement to be able to write modules of code in machine language (assembly language). This requirement exists so that one can meet the demands of a very efficient use of such a computer system. Experience with SAMSO projects in the past indicate that for most applications the amount of such code should be under 15 percent of the total amount of code produced. However, it is often a very crucial 5 to 10 percent of the total amount of code. In the past, this requirement has been satisfied by allowing the programmer to "step down" to assembly language in the middle of a module of code written in a higher order language. The requirement for this form of linkage was dictated by the need for efficient execution time code. The linkage of code modules written in a higher order language to those written in assembly language can be made in other ways. Because of the need for clarity, transferability, and maintainability in any Air Force standard language, there should be no capability to "step down" within a module of code. This requirement for linkage will be met in one of the following ways:

- Direct linking of two independently compiled or assembled subroutines,
- The inclusion of open routines that are either a part of the system library or a part of the user-defined library.

5.3 IMPLEMENTATION CONSTRUCTS

This section complements Section 5.2 in that it contains an outline of the requirements that should be reflected in any implementation of the language constructs and supporting software environment. The section contains material relative to (1) implementation philosophy for any particular compiler, (2) the compile time environment, and (3) the run time environment. Again, the intent is to provide a stable and standard environment based upon well developed software constructs.

5.3.1 Implementation for Any Compiler

Any compiler for the language must implement only the language specified and may not expand on the language specifications. Further, any implementation must implement the entire language; there will be no proper subsets. However, taking into account the hardware limitations of some of the small computers, a compiler for a particular machine need not run on that machine, i.e., self-hosting is not required. Further, a particular compiler may support the efficient use of only a portion of the language features. In particular, in such a situation the compiler will produce warning messages to indicate when features of the language are being used that result in inefficient code. When this feature is combined with "automated coding standard enforcers," the desired operational effect of language subsets can be achieved in practice. The compiler should have the capability of producing optimized code. For example, the compiler will evaluate constant expressions at compile time. Depending upon the particular implementation, the facility may exist such that the user can determine whether speed or memory utilization is to be optimized. To the maximum extent possible, the compiler and the supporting software aids for any particular implementation will be written in the language.

5.3.2 Compile Time Environment

The language and compile time environment will support a number of types of compiler directives. These include debugging facilities, code modifications, and the ability to obtain statistical information about the behavior of the program. The dynamic debugging facilities will be given in terms of the source language and will include commands to set and reset break points, to set variables, and to return control to the user in the event of a program error. The language will provide compile-time-executable statements which will include (1) the ability to modify and/or augment sections of the code and (2) to generate code for run time checking of variables. A uniform set of diagnostic messages will be described that will enable the user to rapidly pinpoint the source of his syntactical error. Where feasible, types of variables, expressions, and parameters will be checked for compatibility across separate modules of code.

5.3.3 Run Time Environment

The run time environment will support the debugging facilities described above. In addition, when requested, it will support the range checking features described in the compile time environment. Run time errors will be reported in terms of the structure of the source program and there will be a standard set of diagnostic messages. Parameters between code modules will be checked at loading and must agree on type and size. There will be a standard facility to describe and support the structure of a program. This will include both the overlay of code and the overlay of data.

The user will be able to cause separately compiled modules of both program and data to be inserted into his program. Further, such insertions of programs may be written in a different language. There will be a facility to support both system and user libraries. There will be a standard method of interfacing with the system, i.e., the job control language and associated control statements. To facilitate the exchange of programs and data between different computers, there will be a common data format for transferring records between machines.

5.4 COMPILER AND OPERATING SYSTEM TESTERS/QUALIFIERS

To ensure that the language and environment standards are followed it is necessary to have a means of checking any particular implementation to determine how closely it meets such standards. Since the experience of software developers over the past 15 years indicates that such "certification" is not easily achieved, it is necessary to have a series of software support tools to aid in this qualification process. Such tools include an extensive set of test cases and supporting software aids. The test cases should fall into one of the following three classes:

- Main-line cases that test the commonly encountered language construct forms and operating system conditions. These should include numerous small cases that test one or more specific constructs of the language as well as some large computer programs.
- Barely "legitimate" cases that probe the edge of the specifications for both the language and the software development environment.
- Barely "illegal" cases that probe the edge of the specifications ensuring that invalid inputs raise proper diagnostic messages.

5.5 CODING STANDARD ENFORCERS

Although higher order languages, structured programming, and other comparable developments in software technology provide much assistance in making individual pieces of coding intelligible, it is still very easy to create confusing code. One technique to further add clarity to code and prevent undesirable programming practices is the use of automated coding standard enforcers. The function of such an enforcer is to prevent coding techniques or particular language constructs from being used in a particular application. Although the language constructs delineated in Section 5.2 provide support for "structured programming," they in no way create an environment in which a programmer must produce structured code. If a particular project office wishes to insist that the programs be written using structured coding techniques, such enforcement should be done through the use of an automated coding standard enforcer. Another requirement for such enforcers arises when, because of hardware or other constraints, a program office decides not to use certain features of the language. Using coding standard enforcers, one can obtain the desirable effects of subsets of a language without sacrificing transportability of code or other economies of a single standard language.

5.6 RELATIONSHIP BETWEEN SOFTWARE DEVELOPMENT ENVIRONMENT REQUIREMENTS AND SOFTWARE MANAGEMENT CONTROLS

AFM 800-14 (Volume II) presents a description of the planning, processes, and procedures used in the acquisition of computer hardware and software to meet data processing requirements in systems procured by the United States Air Force. Early in the development phase of the system acquisition cycle, computer resource requirements are established. There is a strong undertone in the reading of this manual (AFM 800-14) that a well-stocked library of standard software tools exists. As of the date of this report, this is not the case. This report is an effort to describe the requirements for such standard software tools. Developed against such requirements, these software support tools would then become members of the library against which the management directives of AFM 800-14 can then be followed.

6. SAMSO/AIR FORCE REQUIREMENTS FOR CONTROL OF SOFTWARE DEVELOPMENT

The compiler and the associated support software tools necessary to round out the software development environment are complex and sophisticated computer programs. The thrust of the effort to standardize on a higher order language and software development environment for SAMSO computer programs is motivated by anticipated cost savings and reduction in elapsed time necessary to implement a particular operational computer program. If these two goals are to be achieved, it is necessary that not only standards be established, but that these standards be enforced and that software support tools be made available across the boundaries of SAMSO/Air Force operational programs. For example, if the methodology of compiler implementation is not carefully controlled, the compiler can introduce into the system both semantic ambiguities in the interpretation of the higher order language and a myriad of distinct, nonstandard mappings to the languages of the various target machines, thereby defeating the purpose of a single higher order language. Also, as indicated in Appendix B, SAMSO/Aerospace has developed a compiler writing technology that should be used in the implementation of any particular compiler. To achieve these goals it is necessary for SAMSO to maintain control of all phases of the implementation and maintenance of any particular compiler and its associated software development environment. This means that a group should be established to act as a focal point for such a development. The Air Force should own the proprietary rights for the compiler writing system used to develop any particular compiler and should be able to provide this system and any previously developed compiler or other software support tool as Government Furnished Equipment to any software contractor. To ensure that the standards for both language and software development environment are met, this group would test and certify any particular implementation. This group would also be the focal point for all maintenance activities, for the receipt of trouble reports, the issuance of quick fixes and periodic updates.

7. RECOMMENDATIONS

Based upon the analysis given above, it is technically feasible for the vast majority of all computer programming for new SAMSO projects to be done in a higher order language. When this technical feasibility is combined with the increasing cost of software development and maintenance, standardization on a single higher order language becomes highly desirable. However, language standardization is only part of the issue, if cost savings in software development and maintenance are to be realized. The other main features that must be attacked and successfully resolved are the creation of a standard software development environment and the establishment of a program for the specifications, development, test, and maintenance of this single higher order language and its associated software development environment. The above recommendation should not be construed to imply that all programming can be done in a higher order language. As long as there are hardware limitations that are manifested as restrictions in memory size and instruction execution time, there will be a need for code written in assembly language. However, such modules of code should be written only in exceptional circumstances and should be organized as independent subroutines or subprograms. Also, the interfaces between programs written in assembly language and those written in a higher order language should be well defined as a part of the software development environment.

The language specifications that are necessary within any higher order language for SAMSO projects have been outlined above. Throughout the SAMSO software community there is general agreement on the requirements for software constructs necessary for (1) data declaration, (2) data grouping, (3) imperative instructions, (4) control statements, (5) program structure, and (6) input/output. Problems often arise in language specifications when decisions are being made as to the degree of sophistication required. The language must be rich enough to handle a wide range of applications but should not be so expansive and general that it becomes hard to understand, implement, and use. The experience obtained through the use of JOVIAL, FORTRAN, and SPL should be taken into account in the actual specification process. The specifications should be a blend of sufficiently rich constructs that support a broad spectrum of applications.

Of equal importance to the development of a set of specifications for a higher order language is the development of a set of specifications for the software development environment. Such things as debugging facilities, overlay structure of programs, and the control philosophy of the operating system must be specified and standardized, if indeed the economies of language standardization are to be realized.

As a higher order language is being specified it is important to realize that just as software for a flight or ground system has a life cycle, a higher order language will also follow a similar pattern. The initial preparation of specifications for such a higher order language is only the first step in such a life cycle. If successful language standardization is to be achieved, it is necessary that there be a 2- to 5-year specification, development, test, and evaluation effort before the use of the higher order language can be mandated for all new SAMSO programs. Such a language should be specified and the specifications reviewed by each SAMSO program office with experience in software development. This should be followed by a pilot implementation and use by a few SAMSO program offices so that the language constructs and software development environment can be evaluated and refined prior to final specifications. After the pilot program, standards, and final specifications can be established and implementation can begin.

Two essential elements of this recommendation are the need for a higher order programming language for new SAMSO projects and the need for standardization in the software development environment. In addition, to ensure the maintenance and implementation of these specifications across all SAMSO programs, a group should be created with the responsibility for the implementation, development, verification, and certification of all language processors and related software support tools that form the software development environment. This group would also be the focal point for all maintenance activities, for the receipt of trouble reports, the issuance of quick fixes, and periodic updates.

8. POSTSCRIPT

Software is continuing to increase as an important subsystem component of all large weapon systems. The costs to produce reliable software are continuing to rise concurrently with a continuing increase in software complexity. An estimate of the yearly expenditures on software by the Air Force is \$1.5 billion per year.[1] It is reasonable to assume that \$500 million per year of this cost is for weapon system software. If the current trend continues, by 1980 this weapon system software cost will be well over \$700 million per year. On the average, 20 percent of the software costs of any software project are for software support tools, such as language translators, operating systems, and library support. By using a single higher order language and standardizing on the software development environment, this 20 percent cost estimate should be reduced to 5 to 10 percent. Other cost savings should be achievable as secondary efforts. For example, (1) it will not be necessary for a new project to specify a language with the resultant slippages in the project schedule as design errors in the language specification impact the application development, (2) standardization in the language and software development environment will permit sharing of application programs, and (3) once the single higher order language has been used for a few years, programmer training costs will be substantially reduced.
Table VI gives the yearly savings for weapon system software costs after the standard has been established and the first implementation made. The measure of operational life expectancy begins with the first implementation of the standard. For example, if work on the standard is assumed to have begun immediately and took 5 years to design, pilot test, and establish, the year 2000 must arrive before a life expectancy of 20 years can be reached. A life expectancy for such a language of 20 to 40 years is not unreasonable. After all, FORTRAN is 15 years old and going strong.

[1] Information Processing/Data Automation, Implications of Air Force Command and Control Requirements in the 1980's (CCIP-85): "Highlights," Volume I, SAMSO/XRS-71-1 (Los Angeles, Calif.: SAMSO, Los Angeles Air Force Station, April 1972) (Page 4).

Table VI. Life Expectancy (Years)

$ (Billion) Savings	10	20	30
	1.05	2.1	3.15

Unfortunately, one cannot wave a magic wand and achieve standardization and the resultant cost savings. However technically feasible and desirable the goals of a single higher order language may be, the transition period from the environment of today to a single higher order language environment is a lengthy and difficult one. For example, when this report was reviewed by System Program Offices within SAMSO they emphasized their present and future reliance on commercially supported languages. Several advocated that any required DOD HOL features be developed as extensions of a commercially supported language, preferably FORTRAN. The most significant arguments in favor of this position are (1) large investment in proven programs and programming tools, (2) availability of trained personnel within SAMSO and particularly within contractor organizations, and (3) the ability to directly use future software support developed by commercial organizations and the likelihood that machines will be optimized with regard to implementation of commercial languages.

On the other hand, several SAMSO organizations are heavily dependent on JOVIAL J4 and would, of course, like to see any new language be as compatible as possible with that dialect. These System Program Offices also make use of the COMPOOL feature and will be in great difficulty if that is not implemented in any new DOD language development.

Further, such standardization on a single higher order language and associated software development environment will not eliminate many of the difficulties of producing reliable software for complex weapon systems.

Further, unless (1) the single higher order language is a good language in which to program, (2) implementations are carefully certified and high standards maintained, and (3) a single group is given responsibility for the maintenance and extension of the language, the whole effort towards standardization will be, at best, an expensive exercise in futility.

The following questions need to be answered before major steps are taken along the road to standardization:

- Will the directive requiring use of a single higher order language and its associated software development environment have sufficient backing such that it will be followed in practice?
- Will the group responsible for the maintenance and extension of the language be motivated to establish an active user group to encourage feedback from the users to the implementers and maintainers?
- Will there be time for specification, pilot implementation, and initial shakedown of the new language before its use is made mandatory?
- Will the development program and continuing maintenance program be adequately funded?
- How would the code written under this new higher order language interface with that vast body of FORTRAN code currently used and being extended under contractors for the Department of Defense?
- The existence of a standard language and software development environment for weapon systems will be much easier to accomplish and may result in defacto standardization of computer hardware. Is such standardization desirable?

APPENDIX A. SAMSO FUNCTIONAL REQUIREMENTS FOR COMPUTER PROGRAMS*

A.1 INTRODUCTION

The purpose of this appendix is to provide a brief overview of the SAMSO operational programs that require computer program support. This overview is provided by listing the generic functions and by dividing such functions into (1) mission planning and technology development, (2) mission development and mission flight, and (3) mission support.

A.2 FUNCTIONAL REQUIREMENTS

A.2.1 Mission Planning/Technology Development

The type of activity done under mission planning and technology development is primarily oriented to research and development, determination of mission profiles, and a wide variety of different types of simulations. By and large, the type of software development can be predominantly characterized as programming in support of engineering tasks. For example, in the development of a mission profile, it is necessary to do targeting and range safety studies and generate the engineering details and parameters for a flight program. Typical simulations include studies of the effectiveness of a weapon system, analysis of sensor systems and their impact upon command control systems and engineering simulations of specific control systems for a particular missile. Technology developments require a broad range of engineering support software. Structural analysis (matrix manipulation) and fluid mechanics (solution of differential equations) codes are two examples.

A.2.2 Mission Development and Mission Flight

The software required to support mission development functions must be subject to extremely thorough verification and validation. Although

*This appendix was prepared by Major Carl Jund, SAMSO/DYAC.

this requirement is not explicitly delineated in any of the subsections below, the reader should at all times keep in mind the need for very reliable software. The following subsections represent most of the separable SAMSO mission functions.

A.2.2.1 Prelaunch Checkout

The prelaunch and checkout function usually consists of the development of verification runs that verify the operability of equipment and its status. Sensors and actuators are exercised/calibrated to determine if the operational limits have not been exceeded. A trial scenario including retargeting may be exercised and equipment anomalies recorded. Considerable input-output processing is necessary to support this function. This activity also includes automatic test and evaluation functions.

A.2.2.2 Command and Control and Supporting Displays

The software functions necessary to support a command and control application usually involve the transmittal and receipt of large amounts of data at high data rates. Since the data must be made available to decision makers if it is to be of any value, it is necessary to support command and control systems with computer controlled graphic displays. Software for such displays usually requires considerable array and character manipulation activity. If the data must be transmitted over a nonsecure link, it may be necessary to use cryptographic devices. Economical use of data transmission capabilities almost always requires encoding and decoding of information. The functional requirements of a command and control system demand that the system act in an asynchronous mode, and that it be able to support interrupts received both from the decision makers and the external world.

A.2.2.3 Navigation, Guidance, and Control

Guidance and control of the vehicle involves calculations for determining vehicle attitude and thruster on and off times. Calculations made for navigation determine and predict position and velocity vectors. These usually involve matrix calculations, coordinate conversions in three dimensions, and the solution of ordinary differential equations.

A.2.2.4 Surveillance and Reconnaissance

The software functions necessary to support surveillance and reconnaissance involve the receipt, storage, and transmission of large blocks of data in a wide variety of data formats. Often it is necessary to do on-board preliminary data processing to reduce the data volume. Surveillance requires identification of targets and considerable decision making, data analysis, and statistical evaluation. This task is largely one of primitive pattern recognition. The reconnaissance task is similar to the task of surveillance although more sensor information is being processed, stored, and transmitted.

A.2.2.5 Weapon Delivery

The functional software task involved in weapon delivery requires the real-time transfer and verification of critical parameters from the local control center to the weapon computer. Considerable sophistication in the software is required since the task must meet extremely strict security requirements.

A.2.2.6 Space Payloads and Experiments

The functional software tasks required to support experiments in space include command and control activities, data acquisition, reduction, and transmission.

A.2.2.7 Man Rating

Man rating is a task of aiding the astronauts in the analysis and monitoring of physical health and fitness. The functional software required for this task supports the monitoring, data acquisition, data reduction, and transmission activities. Because of the concept of man rating, the requirements for verification and validation of the software to ensure flight safety requirements are extremely high.

A. 2. 3 Mission Support

The software necessary to support this phase of an operational program consists of simulation of vehicle, simulation of a flight computer, and postflight data reduction and analysis.

A. 2. 3. 1 Vehicle Simulation

Vehicle simulation is a tool which is used to ensure that the vehicle will perform as predicted during its flight. Such simulation requires a broad range of software support tasks, such as the development of mathematical relationships and models, data processing and processing of command and control messages.

A. 2. 3. 2 Computer Simulation

As in vehicle simulation, simulation of an on-board flight computer is a means of ensuring its performance. Usually this is done by an interpretive simulation of the flight computer within a large general-purpose computer. The software tasks required to support such a simulation are extremely varied.

A. 2. 3. 3 Postflight Data Reduction and Analysis

Once a mission has been flown, it is necessary to reduce and analyze the data obtained from the mission in order to determine how well the objectives of the mission were achieved. The software tasks required are data formatting, data manipulation, data structuring, and mathematical modeling.

APPENDIX B. COMPILER WRITING TECHNOLOGY ASSESSMENT

The Air Force is interested in software systems which can facilitate the production of higher order programming language compilers for the minicomputers used in space, missile, and avionics systems. "A higher order language (HOL) is a relatively machine-independent computer language which uses English words and statements where they are convenient, combined with mathematical notation, to express algorithmic procedures."[2] A compiler is the computer program that translates from the programmer's HOL into the language which activates the digital logic of the computer. The compilers developed for these real time processors must satisfy stringent requirements for the production of executable code that is highly efficient both in memory utilization and in execution speed. The methodology must also be cost effective since the wide variety of digital processors available for weapons systems applications requires a multiplicity of compilers.

Before assessing the technology, we must understand the problem. Basic information about compilers and compiler writing tools will be presented first, followed by an overview of the state of the art and a discussion of the SAMSO experience within one particular system.

B.1 COMPILERS

A compiler is a translator program that either transforms an HOL program into an assembly language form (i.e., mnemonics for machine code) for subsequent assembly to machine language translation by the assembler, or which transforms directly the HOL program into an equivalent machine language program.[2]

Compilers may be described as being interpretive or generative, syntax directed, or nonsyntax directed. In interpretive type systems, the

---

[2] Christine M. Anderson, Aerospace Higher Order Language Processing, Report No. AFAL-TR-73-151 [Wright Patterson Air Force Base, Ohio 45433, Air Force Avionics Laboratory (AFAC/AAM), June 1973].

syntactic structure of each source language statement is determined and immediately executed one at a time. No object code is generated. In generative compiler systems, the entire program is analyzed and object code is generated for subsequent execution.[2]

Syntax directed compilers make use of formal syntax descriptive languages in parsing the source language statements. An example of such a language is Backus Naur Form (BNF). Compilers written with the aid of syntax descriptive languages foster relative ease in modification of the grammar.[2]

In contrast, nonsyntax directed compilers proceed in an ad hoc fashion, attempting to recognize each statement type and structure as it is encountered with no formal rules to follow. Language modification in such systems is difficult.[2]

The analysis phase, often referred to as the "front end" of a compiler, performs two functions: lexical and syntactic analysis. The lexical analysis is the simplest part. The input to the compiler is a string of symbols from an alphabet. The lexical analyzer must group together certain characters into single syntactic entities, called tokens. What defines a token is established by the specifications of the programming language. The output of the lexical analyzer, a stream of tokens, is the input to the syntax analyzer.

The syntax analyzer, often called a parser, matches the syntax definition with the symbols of the program. It builds an internal form, perhaps a tree structure, of the program after disassembling the structure of the source program.

The semantic analyzer is called to check for semantic correctness and to modify and expand the trees into a more complete form. Frequently, at this point, information about the variables is processed. This includes the building of a symbol table and allocation of data. A diagram of a compiler is shown in Figure B-1.

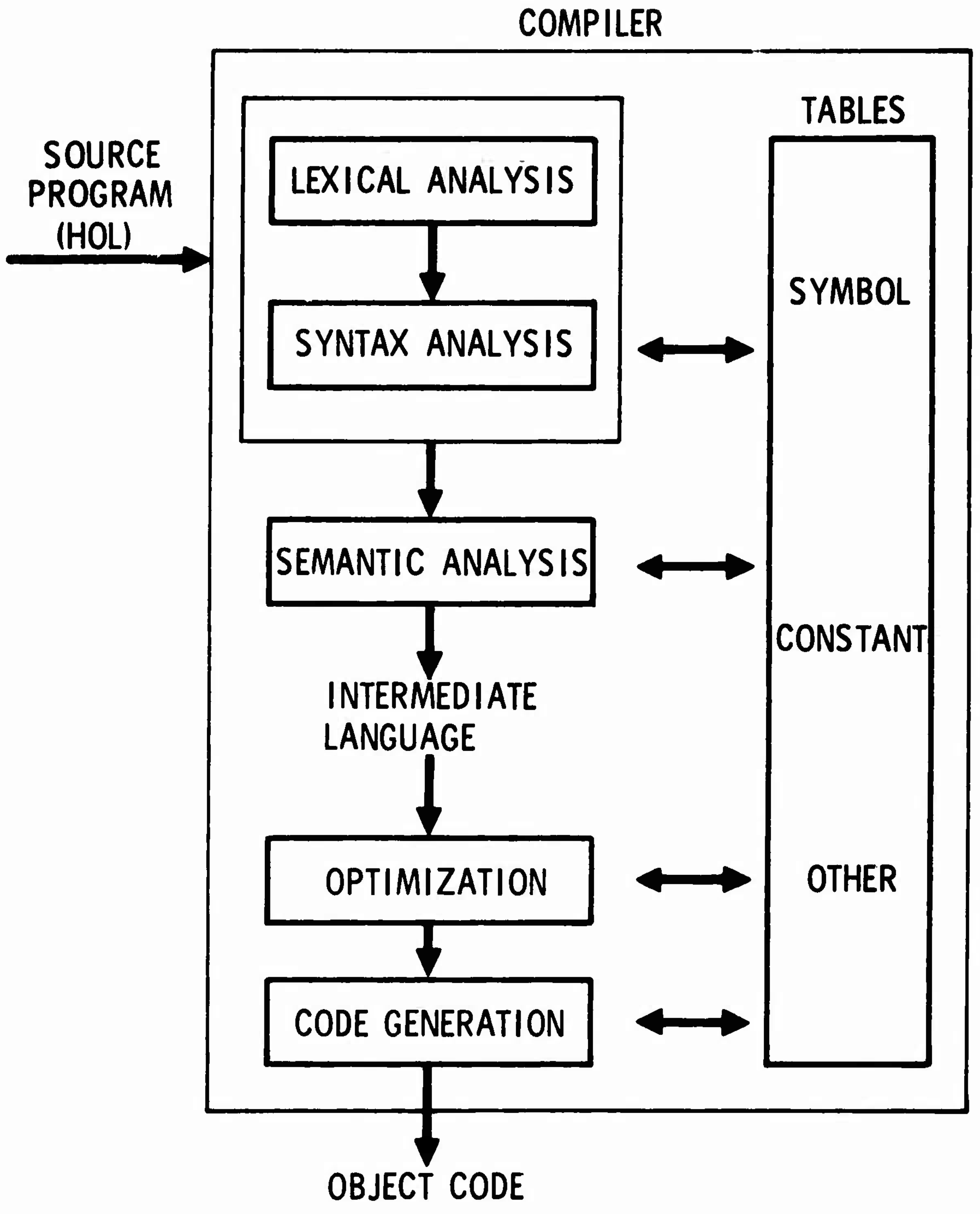

Figure B-1. A Compiler[1]

Code optimization is the attempt to make object programs more efficient, in the sense of faster running or more compact. The complexity of the optimization process depends upon the source language and the desired efficiency of the generated code. This phase is largely machine independent. In practice, one must be content with code improvement, encompassing such functions as: elimination of common subexpressions, removal of unnecessary inner loop computations, detection of statements that will never be executed, and constant propagation.

The code generation phase or "back end" of a compiler is the actual translator of the internal form into assembly or machine code. From a compiler writer's viewpoint this portion is straightforward but time consuming. This phase, unlike the others, is heavily machine-dependent. Such considerations as the number of registers and I/O interface must be taken into account.[2]

The complexity of the language to be implemented impacts heavily on this phase. For example, code generation for full FORTRAN costs about 25 percent of the total compiler cost. However, for a much larger language like PL/1, code generation involves anywhere from 45 to 75 percent of the total cost.[2]

B.2 COMPILER WRITING TOOLS

Much attention has been given in recent years to methods of automating the development of compilers. This research has produced a variety of software systems referred to as metacompilers or compiler-compilers. Ideally, such a system should be capable of accepting a formal description of a language and produce from it a set of tables for driving a skeleton compiler which is language independent. This ideal has been attained in part. Automation of the front end has met with the greatest success because it is more machine independent. More research is needed in both formalization and automation of the optimization and code generation phases. Current metacompiler systems provide useful tools, but only when in the hands of experienced compiler writing professionals.

Basic to the understanding of metacompilers is the concept of metalanguage — a language used to describe another language. A metalanguage can describe the syntax, semantics, or target machine characteristics in terms which are general enough to define different languages on different machines. "Experience shows that languages specifically designed for compiler development approach 90 percent of the efficiency associated with assembly language coded compilers. Expected advances in optimization could make metalanguages standard tools for compiler production within the next few years."[2]

A metacompiler translates a metalanguage into code which can be understood by the machine. Metacompilers can usually build part of the compiler based on information supplied by the programmer's metalanguage code.

In attempting to automate compiler writing, metacompiler designers have been forced to isolate the distinct phases of the compiler, resulting in a modular program design. Modularity facilitates modification. In systems such as SPLIT, one can re-metacompile portions of the compiler without having to redefine all parts of the compiler.

Consider the following examples of frequently desired compiler modifications, demonstrating the effect of modularization on compiler generation. First, let us define two more terms. Host refers to the computer on which a compiler executes. Target refers to the computer for which object code is being generated.

(1) Rehosting the metacompiler. Usually the generated compiler will run only on the host machine. "Rehosting requires changing all portions of the metacompiler that are host dependent."[2] Rehosting can be a large job if many of the support routines are coded in a host dependent language.

(2) Changing the HOL. Suppose a certain compiler can translate from language A to language B and it is desired to translate from language C into language B. This modification requires recoding the front end of the

existing compiler to allow for the syntax and semantics of the new language. The code generation portion of the compiler may not require modification.

(3) Retargeting the compiler. This modification requires that the code generator portion of the compiler be recoded to produce code which will execute on a new target computer. The front end of the system remains the same.

B. 3 STATE OF THE ART IN METACOMPILER TOOLS

Many companies and universities have developed significant compiler writing tools.

- SPLIT

  Space Programming Language Implementation Tool (SPLIT) was developed by Systems Development Corporation (SDC). It is a syntax directed metacompiler designed specifically for the generation of SPL (Space Programming Language) compilers. This system is owned by the Air Force and is available to the public.

- CWIC

  A Compiler for Writing and Implementing Compilers (CWIC), a forerunner of SPLIT, is an SDC proprietary system of tools for constructing compilers, interpreters, report generators, data base generators, and editors on the IBM 360.

- GENESIS

  GENESIS is a proprietary compiler writing tool developed by Computer Sciences Corporation (CSC). GENESIS focuses primarily on the generation of the front end of the compiler. It accepts as input the formalized description of a programming language.

- JOCIT

  CSC developed this system under contract to RADC. JOCIT (JOVIAL Compiler Implementation Tool) is designed specifically for the JOVIAL dialect J3. JOCIT isolates host and target machine dependencies to facilitate the rehosting and retargeting of J3 compilers.

- XPL

  This system was developed by university people from University of California, University of Toronto, and Stanford University. XPL features semiautomated front end generation, accepting input in BNF metalanguage. The XPL language is a dialect of PL/1 and runs on the IBM 360.

- AED

  The Automated Engineering Design (AED) system is a software system used to build compilers, operating systems, computer graphics, data management systems, and large application systems. It was developed by the MIT Computer-Aided Design Project funded by the Air Force Materials Laboratory. Public AED is being supported by SofTech.

B.4 OTHER APPROACHES TO COMPILER WRITING

"In addition to providing special languages with which to write compilers and attempting to automate the production of various portions of the compiler, there are two other approaches to compiler writing that may be used in conjunction with the first two techniques."[2]

The first approach is known as cross compiling and consists of writing a compiler which will run on a host machine and which will generate code to run on another machine. Univac has employed this technique a great deal; for instance, in developing a compiler for the AM/UYK-7.

The other is development of a multi front and back end compiler writing system. This approach features the concept of intermediate language (IL) standardization across many source languages. This results in a high degree of modularization which facilitates front and back end modification to produce a variety of compilers.[2]

Little work has been done in this area, primarily due to the fact that the requirement for parameterized off-the-shelf compilers for target machines is minimal in a commercial environment. Differences in architecture among avionic and space computers which are significant to compiler optimization and code generation include: (1) logic (hardwired versus read only memory); (2) registers versus accumulators; (3) instruction format; and (4) input/output interfaces.[2] An example of intermediate language standardization is shown in Figure B-2.

B.5 SAMSO EXPERIENCE

The Aerospace Corporation, under funding from SAMSO, has been actively involved for the past five years in the application of the Air Force-owned SPLIT (Space Programming Language Implementation Tool) system to facilitate the production of HOL compilers for avionics minicomputers.

The basic SPLIT system was used successfully to develop SPL compilers that generate executable code for the Honeywell DDP-516 and RCA SCP-234 computers. The first was used to implement the flight program and ground laboratory support software required for the Space Precision Attitude Reference System; the second to implement the flight program for the Defense Meteorological Satellite. Each compiler was produced in 10 man months, delivered in five calendar months, and thereby demonstrated the effectiveness of the methodology in meeting all efficiency goals in a cost-effective manner.

Following completion of these programs, The Aerospace Corporation began a 2-year program sponsored by the Deputy for Technology of the Space and Missile Systems Organization to improve and enhance the basic system. Because no standard programming language currently exists for the general class of real-time, computational systems, The Aerospace Corporation felt it was important that the system be enhanced and expanded to readily accept any of the various languages now in use for such systems to minimize the costs of its use. It was also decided to change the name of the system to Compiler Writing System, to reflect more accurately its potential use in the standardization both of programming languages and of their implementation.

These efforts led directly to the interest of the Langley Research Center of the National Aeronautics and Space Administration that has resulted in the transfer of the Compiler Writing System to Langley for their use in the

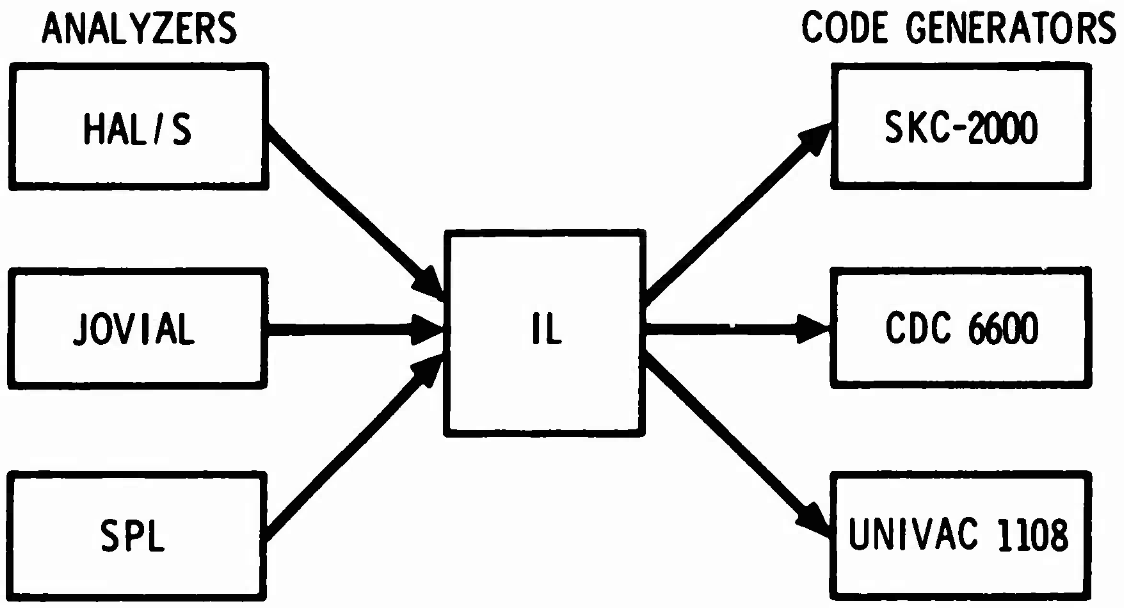

Figure B-2. Intermediate Language Standardization production of compilers for a wide variety of flight computers. They estimated the initial cost savings to the Government from the use of the Compiler Writing System to be in excess of $750,000.

B.6 ASSESSMENT

Compiler writing systems have a substantial cost advantage over traditional methods of compiler production. The advantages are overwhelming if the establishment of a standard language and the desirability of producing executable code for a variety of avionics and space computers for various phases of mission requirements is assumed.

It is essential that the language specification must produce a consistent, unambiguous, context-free grammar. If the language is defined in a syntax-directed manner (that is, not dependent on the language but on how the language is described), then the production of the front end of a compiler is straightforward. In fact, it is estimated that if one started with a reasonable language specified in Backus-Naur Form with an LR(1)[3] structure assured, a syntax analyzer could be produced in a single man-month. Difficulties with the front end are always due to inadequate or ambiguous specification of the grammar.

A compiler writing system helps in the development of the language itself — and if so used, can be helpful in the verification and validation of both the language design and of the compiler itself. The verification and validation advantage represents a significant gain not offered by any traditional method.

There is virtually no penalty for using a compiler writing system. The development will not take longer and will probably take less time than the usual brute force methods. If two programmers code from the same flow diagrams, one in assembly language and one using metalanguage, the code

[3] Alfred V. Aho and Jeffrey D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. I:"Parsing", Vol. II:"Compiling"(Englewood Cliffs, New Jersey: Prentice Hall, 1972) (p. 371).

produced using the metalanguage will be better than 85 and close to 90 percent as efficient, in terms of both speed and time, as the direct code.

Once a standard language is defined, what remains is the problem of writing back ends for compilers for multiple target machines. The cost for a compiler for a new avionics machine is then equal to the cost of the back end alone. It is estimated that two people can produce a code generator in 6 elapsed months, even allowing for the study of the target machine architecture and for system implementation problems. For similar architecture, as in families of computers, modification to the back end will be minor and cost will be minimal. The costs of development are strictly in the machine dependent portions of the code.

Even greater advantages are to be gained if further modularization and parameterization can be performed on the back end, particularly in the isolation of all machine dependent portions. This process is similar to what has already been accomplished for the front end analyzers.

One significant advantage of such systems is that as additional compilers are produced, more cost savings are realized. Additional compilers, because of retargeting or rehosting, always cost much less than the original product.

Consideration should also be given to the influence of the system on project management. First, in establishing one HOL, management has complete control of the language. All language-related development, the most expensive area, need be done only once. Any improvements to the front end made by any user can be easily shared with all users at other installations because of the ease of generating code after having previously rehosted the whole compiler tool. Also, subroutines in a library, or support package, written in the HOL, or metalanguage, can be made available to any user. Certainly the implications to costs savings are clear in an environment in which many contractors will be involved in all the phases of mission analysis, test bed generation, ground and space software production, and in verification and validation procedures.

APPENDIX C. OUTLINE OF PLAN FOR SOFTWARE DEVELOPMENT REQUIREMENTS

C.1 INTRODUCTION

This appendix presents a very brief outline of a program plan. The purpose of this program would be for the development, implementation, test, and maintenance of a higher order language and a software development environment to support all SAMSO computer programming for SAMSO operational programs. The experience of knowledgeable personnel within SAMSO/Aerospace indicates that such a language and the associated plan could support all Air Force requirements for computer programming for operational Air Force Programs. This occurs because the requirements imposed on computer programming by SAMSO operational programs are sufficiently broad. The plan is concerned with the steps necessary to prepare specifications, continue development of compiler writing aids and other software support tools, and to maintain an assemblage of such software support tools. The requirements delineated in Section 6 of the above report make it mandatory that the Air Force own the proprietary rights, not only for any particular implementation of the language and software development environment specifications, but also for the compiler writing system and other support tools. This should be done to reduce the total costs that the Air Force must pay in the development of software for its operational programs and should also be done to ensure that no one software contractor obtains an unfair competitive position with respect to particular procurements for individual implementation of the language and software development specifications.

C.2 DEVELOPMENT PLAN

The development plan consists of two parallel and interacting activities. The first is the development of the specifications and associated implementations. The second is the continuing support and extensions of the software technology that supports the implementation of particular compilers and other software support tools. As indicated in Figure C-1, these activities would be done in a parallel manner.

The first step that must be taken in the sequence of specification preparation through implementation of a single higher order language is the translation of the brief requirements delineated in the body of this report into a set of preliminary specifications containing both the syntax and semantics of the new language and execution environment. The group that is responsible for these preliminary specifications should be composed of individuals well versed in language development and particular applications of SAMSO computer programs. Once the preliminary language specifications and software development environment specifications have been prepared, it is necessary that they be reviewed by different program offices within SAMSO that have considerable software experience. In addition to this review before final specifications are prepared, it is essential that there be a pilot implementation of the complete language and software development environment. This should be done for at least two medium to large scale computation systems that are widely used within the Air Force. The requirement for such a pilot implementation stems from the experiences of personnel in the development and implementation of higher order languages and operating systems. Even when great care has been takenwith the specification preparation, flaws, omissions and inconsistencies have always arisen in the actual implementation. The cost impact, when viewed over the expected life of such a single language, is in the billions of dollars. Hence, because of the potential cost impact of such a set of specifications for both the language and software development environment, it is essential that these flaws be minimized. Once the pilot implementations are complete, there must be a period of use and evaluation. Then preparation of the final version of the specification can be accomplished and implemented.

As indicated in Figure C-1, the parallel activity that should be undertaken to support the implementation and maintenance of any such higher order language is the tool sharpening of the associated software technology.

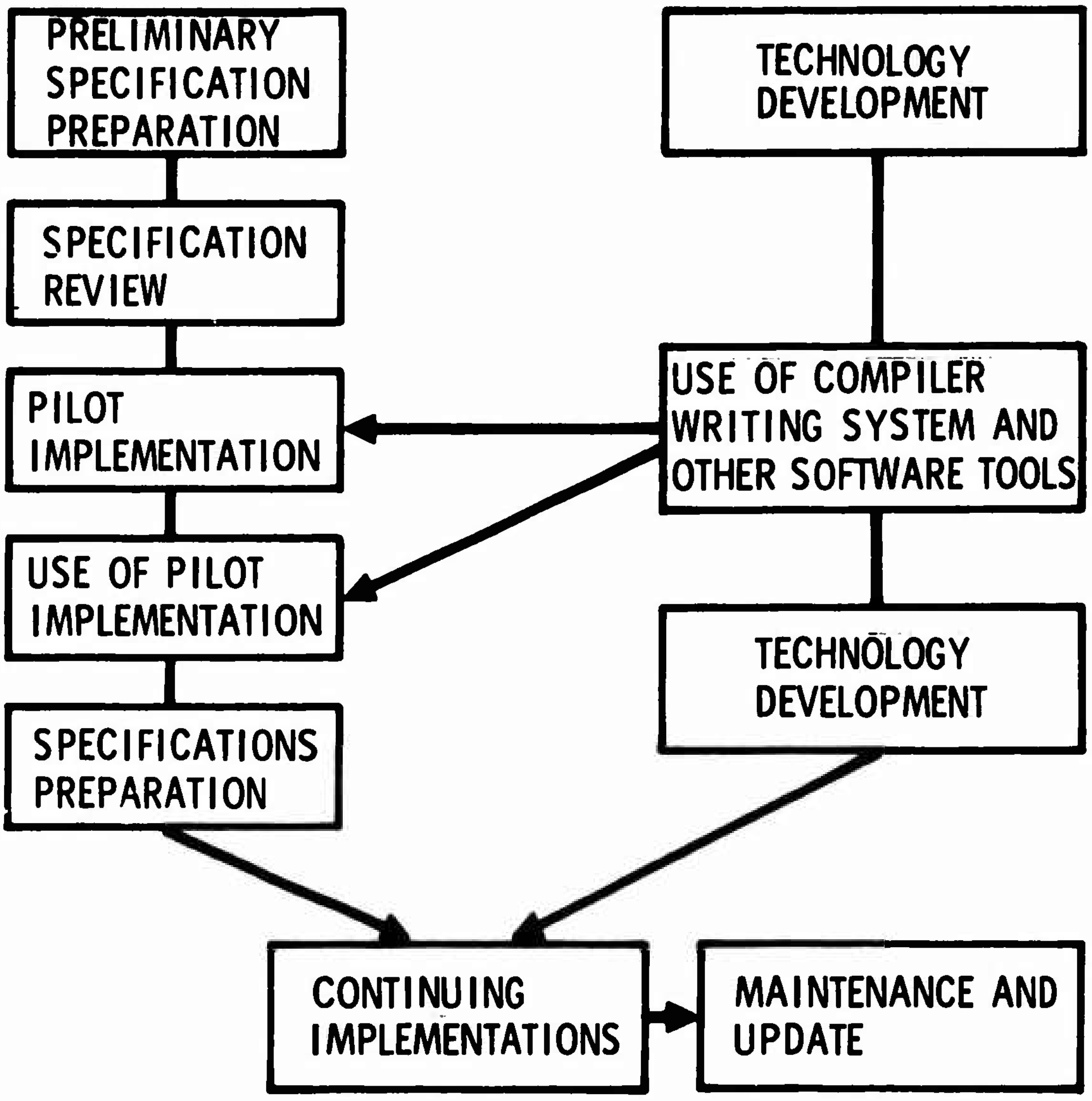
PRELIMINARY SPECIFICATION PREPARATION
SPECIFICATION REVIEW
PILOT IMPLEMENTATION
USE OF PILOT IMPLEMENTATION
SPECIFICATIONS PREPARATION
TECHNOLOGY DEVELOPMENT
USE OF COMPILER WRITING SYSTEM AND OTHER SOFTWARE TOOLS
TECHNOLOGY DEVELOPMENT
CONTINUING IMPLEMENTATIONS
MAINTENANCE AND UPDATE

Figure C-1. Program Plan

This tool sharpening is (1) an extension of the supporting software commonly labeled compiler writing system and other related software tools, and (2) the creation of a group of individuals that would provide technical support on a continuing basis once the final specifications for the language and software development environment have been prepared. SAMSO is in the process of preparing a program management plan for the extension of an existing compiler writing system. The reader is referred to this plan[4] for details of the technical steps that should be taken to enhance the technology necessary to support cost effective implementations of the language and software development environment specifications.

The use of higher order languages for weapons systems software is a relatively new activity in the Air Force and there is a corresponding lack of organic capability to provide the technical expertise required to correctly determine both language requirements and compiler implementation standards. It is also unrealistic to expect that an individual System Program Office would be able to locate and acquire personnel with the necessary software expertise to work directly, and only, for that particular project. As a result, System Program Offices have had to depend on contractors for this expertise and have been unable to obtain support with sufficient objectivity.

A technical support group should be created that would provide support to any Air Force System Program Office in the areas of requirements analysis relating to software, compiler procurement, including technical direction of contractor efforts, acceptance testing of delivered compilers and associated software tools that support the software development environment, receipt of trouble reports, issuance of quick fixes and periodic updates. This group would not be responsible for the specification preparation but would advise those preparing the specifications on the interaction between the compiler writing system, efficiencies of projected compilers and particular specifications of the language. They would be responsible for the

[4]SAMSO/DYAC Program Management Plan, May 1975 (not available outside SAMSO/DYAC)

technical direction of the pilot implementation and would prepare a review of both the specifications and pilot implementation. This review would be used during the preparation of final specifications. Once the final specifications have been prepared, this group then becomes the final arbitrator and judge in terms of maintaining these specifications. The group would be the focal point for all future implementations of the language and software development environment. They would determine whether or not a particular implementation meets the specifications and, hence can be released for operational use. They would also act as a clearing house for trouble reports, issuance of quick fixes, and periodic updates.

C.3 MAINTENANCE

Once final specifications have been prepared and standards established, the activities become primarily maintenance and operation. The technical support group that was created to support the development activities should continue in existence to provide the focal point for the maintenance and operation activities. The services that it would provide to any Air Force System Program Office would be required on an on-going basis. As additional weapon systems are procured that contain computers, there will be the requirement for new implementations. There must, of course, be technical direction of such contractor efforts, verification and certification of the requesting compilers and other software support tools. With a software system as large as this one (standards, specifications, and a large number of implementations), there will undoubtedly be trouble reports and problems that arise during normal use. This group will receive all such trouble reports and will be responsible for the issuance of quick fixes and updates, whether the work is being done within the group or done by software contractors.